FEASIBLE COMPUTATION WITH HIGHER TYPES

HELMUT SCHWICHTENBERG
Mathematisches Institut der Universität München
schwicht@mathematik.uni-muenchen.de

STEPHEN J. BELLANTONI
Department of Computer Science, University of Toronto
sjb@cs.toronto.edu

Abstract. We restrict recursion in finite types so as to characterize the polynomial time computable functions. The restrictions are obtained by enriching the type structure with the formation of types $\rho \to \sigma$ and terms $\lambda \bar{x}^\rho r$ as well as $\rho \multimap \sigma$ and $\lambda x^\rho r$. Here we use two sorts of typed variables: complete ones $\bar{x}^\rho$ and incomplete ones x^ρ.

1. Introduction

Recursion in all finite types was introduced by Hilbert (1925), the system later becoming known as Gödel's system **T** (Gödel, 1958). The value computed by a higher type recursion can be any functional, which is to say a mapping that takes other mappings as arguments and produces a new mapping. Correspondingly one defines a type system of functions and functionals over some ground types.

Recursion in higher types, as in Gödel's system **T**, has long been viewed as a powerful scheme unsuitable for describing small complexity classes such as polynomial time. It is well known that ramification can be used to restrict higher type recursion. However, to characterize the very small class of polynomial-time computable functions while still admitting higher type recursion, it seems that an additional principle is required. By introducing linearity constraints in conjunction with ramified recursion, we characterize polynomial-time computability while admitting recursion in higher types.

We shall work with "recursion on notation", which seems appropriate in the context of poly-time computation. So we consider numbers as represented in binary notation. Recall that every positive integer can then be written uniquely as $1i_1 \dots i_k$ with $i_\nu \in \{0,1\}$, representing $2^k + \sum_{\nu=1}^{k} i_\nu 2^{k-\nu}$. Using the functions $s_0(x) := 2x$ and $s_1(x) := 2x+1$, we may write $1i_1 \dots i_k$ as $s_{i_k}(\dots(s_{i_1}1)\dots)$. In our term language to be introduced below we shall denote s_i by S_i, and the number 1 by $\mathsf{1}$.

We define a restriction **LT** of Gödel's system **T** (Gödel, 1958) such that definable functions are exactly the polynomial time computable ones. To this end we combine

- a liberalized form of linearity for object and assumption variables (allowing multiple use of ground type results) with
- an extension of ramification concepts to all finite types, by allowing

$$\begin{cases} \rho \to \sigma \\ \lambda \bar{x}^\rho r \end{cases} \quad \text{as well as} \quad \begin{cases} \rho \multimap \sigma \\ \lambda x^\rho r \end{cases}$$

  and a corresponding syntactic distinction between incomplete and complete (typed) variables.

This paper grew out of joint work with Karl-Heinz Niggl done in 1998 and reported in (Bellantoni et al., 2000). Its aim is to simplify and clarify some of the concepts involved, with a planned extension (via the Curry-Howard correspondence) to an arithmetical system in mind. In particular, we have changed the constants, simplified the type system and the computation model, and have streamlined the treatment of ramification.

Related work has been done by Martin Hofmann (1998), who obtained similar results with a very different proof technique. Hofmann's recursive system was lifted to a polytime classical modal arithmetic by Bellantoni and Hofmann (to appear). The earlier "intrinsic theories" of Leivant (1995) followed the tradition of quantifier restrictions in induction. Ramification concepts have been considered e.g. by Simmons (1988), Bellantoni/Cook (1992) and Leivant/Marion (1993; to appear); they are extended here to all finite types. Notice also that the "tiered" typed λ-calculi of Leivant and Marion (1993) depend heavily on different representations of data (as words and as Church-like abstraction terms), which is not necessary in the approach developed here. One should also mention bounded linear logic of Girard, Scedrov and Scott (1990), and the so-called light linear logic of Girard (1998). The former differs from what we do here by requiring explicit bounds. A precise relation to the latter still needs to be clarified.

2. Motivation: examples for exponential growth

To set the stage, we discuss some examples of recursively defined functions and functionals exhibiting exponential growth. Our task will be to find appropriate restrictions on types and terms to exclude these definitions.

2.1. TWO RECURSIONS

$$\begin{aligned} d(1) &:= \mathsf{S}_0(1) & \qquad e(1) &:= 1 \\ d(\mathsf{S}_i(x)) &:= \mathsf{S}_0(\mathsf{S}_0(d(x))) & \qquad e(\mathsf{S}_i(x)) &:= d(e(x)) \end{aligned}$$

Then $|d(x)| = 2|x|$, $e(x) = d^{|x|-1}(1)$, i.e. we have exponential growth. The problem is that the previous value $e(x)$ of the second recursion is plugged into the recursion argument of d. Our cure will be to mark recursion arguments (cf. the notions of *safe* vs. *normal* arguments (Simmons, Bellantoni/Cook), and of *tiering* (Leivant)).

2.2. RECURSION WITH PARAMETER SUBSTITUTION

Consider the definition

$$\begin{aligned} e(1,y) &:= \mathsf{S}_0(y) \\ e(\mathsf{S}_i(x),y) &:= e(x,e(x,y)) \end{aligned} \quad \text{or} \quad \begin{aligned} e(1) &:= \mathsf{S}_0 \\ e(\mathsf{S}_i(x)) &:= e(x) \circ e(x) \end{aligned}$$

Then $e(x) = \mathsf{S}_0^{2^{|x|-1}}$. The problem now clearly is that the previous higher type value of the recursion has been used twice. Our cure will be a linearity restriction.

A related phenomenon also involving recursion with parameter substitution occurs if we define

$$\begin{aligned} e(1,y) &:= y \\ e(\mathsf{S}_i(x),y) &:= e(x,d(y)) \end{aligned} \quad \text{or} \quad \begin{aligned} e(1) &:= \mathsf{id} \\ e(\mathsf{S}_i(x)) &:= e(x) \circ d \end{aligned}$$

Then $e(x) = d^{(|x|-1)}$. Now the problem might be localized in the fact that we have a higher result type with argument types "marked" as recursion arguments: the type of the single (recursion) argument of d needs to be the argument type of $e(x)$. Our cure will be to exclude this.

Consider the definition

$$\begin{array}{rcl} I(1,f,y) := y \\ I(\mathsf{S}_i(x),f,y) := f(I(x,f,y)) \end{array} \quad \text{or} \quad \begin{array}{rcl} I(1,f) := \mathsf{id} \\ I(\mathsf{S}_i(x),f) := f \circ I(x,f) \end{array}$$

Then $I(x,f) = f^{|x|-1}$, hence $I(x,d) = d^{|x|-1}$. Now the problem lies in the substitution of the recursively defined d into a function parameter. This again will be excluded by requiring the result types to be without markers.

A related phenomenon occurs in

$$\begin{array}{rcl} e(1) := \mathsf{S}_0 \\ e(\mathsf{S}_i(x)) := I(\mathsf{S}_0(\mathsf{S}_0(1)), e(x)) \end{array}$$

Then: $e(x) = \mathsf{S}_0^{2^{|x|-1}}$. Here the problem is the use of the "incomplete" higher type previous value $e(x)$, occurring as the step argument of I. This will be excluded by requiring that there are no such higher type parameters in the step terms.

3. Types, terms, and denotations

As already mentioned in the introduction, we shall work with two forms of arrow types and abstraction terms:

$$\begin{cases} \rho \to \sigma \\ \lambda \bar{x}^{\rho} r \end{cases} \quad \text{as well as} \quad \begin{cases} \rho \multimap \sigma \\ \lambda x^{\rho} r \end{cases}$$

and a corresponding syntactic distinction between incomplete and complete (typed) variables. The intuition is that a function of type $\rho \to \sigma$ may use its argument many times, whereas a function of the "linear" type $\rho \multimap \sigma$ is only allowed to use it once. As is well known, we then need a corresponding distinction for product types: the ordinary form $\times$ for $\to$, and the tensor product $\otimes$ for the linear arrow $\multimap$. Formally we proceed as follows.

3.1. TYPES

The types are:

$$\rho, \sigma ::= \mathbf{U} \mid \mathbf{B} \mid \mathbf{L}(\rho) \mid \rho \multimap \sigma \mid \rho \to \sigma \mid \rho \otimes \sigma \mid \rho \times \sigma.$$

The level of a type is defined by

$$\begin{array}{ll} l(\mathbf{U}) := l(\mathbf{B}) & := 0 \\ l(\mathbf{L}(\rho)) & := l(\rho) \\ l(\rho \multimap \sigma) := l(\rho \to \sigma) & := \max\{l(\sigma), 1 + l(\rho)\} \\ l(\rho \otimes \sigma) & := \max\{l(\rho), l(\sigma)\} \\ l(\rho \times \sigma) & := \max\{l(\rho), l(\sigma), 1\} \end{array}$$

Ground types are the types of level 0, and a *higher* type is any type of level at least 1. A *function* type is a type of level at most 1. The $\to$-free types are also called *linear* types. In particular, each ground type is linear.

3.2. SET MODEL

There is an obvious set model of our type system, where we interpret every type ρ in the left column by the set $\mathbb{S}^\rho$ given in the right column:

$\mathbf{U}$	a special singleton set
$\mathbf{B}$	a special two-element set
$\mathbf{L}(\rho)$	the set of lists of elements of $\mathbb{S}^\rho$
$\rho \multimap \sigma$ and $\rho \to \sigma$	the set of total functions from $\mathbb{S}^\rho$ to $\mathbb{S}^\sigma$
$\rho \otimes \sigma$ and $\rho \times \sigma$	the cartesian product of $\mathbb{S}^\rho$ and $\mathbb{S}^\sigma$

We abbreviate $\mathbf{N} := \mathbf{L}(\mathbf{U})$ and call it the type of unary numerals; similarly $\mathbf{W} := \mathbf{L}(\mathbf{B})$ is called the type of binary numerals.

3.3. TERMS

The *constant symbols* are listed below, with their types.

$$\begin{array}{ll} \mathsf{xx} & : \mathbf{U} \\ \mathsf{tt} & : \mathbf{B} \\ \mathsf{ff} & : \mathbf{B} \\ \varepsilon_\rho & : \mathbf{L}(\rho) \\ *_\rho & : \rho \multimap \mathbf{L}(\rho) \multimap \mathbf{L}(\rho) \\ \mathsf{if}_\tau & : \mathbf{B} \multimap \tau \times \tau \multimap \tau \quad (\tau \text{ linear}) \\ \boldsymbol{c}^\rho_\tau & : \mathbf{L}(\rho) \multimap \tau \times (\rho \multimap \mathbf{L}(\rho) \multimap \tau) \multimap \tau \quad (\tau \text{ linear}) \\ \mathcal{R}^\rho_\tau & : \mathbf{L}(\rho) \to (\rho \to \mathbf{L}(\rho) \to \tau \multimap \tau) \to \tau \multimap \tau \quad (\rho \text{ ground}, \tau \text{ linear}) \end{array}$$

and for linear ρ, σ, τ

$$
\begin{aligned}
&\otimes^{+}_{\rho\sigma} \;: \rho \multimap \sigma \multimap \rho \otimes \sigma \\
&\otimes^{-}_{\rho\sigma\tau} : \rho \otimes \sigma \multimap (\rho \multimap \sigma \multimap \tau) \multimap \tau \\
&\times^{+}_{\rho\sigma\tau} : (\tau \multimap \rho) \multimap (\tau \multimap \sigma) \multimap \tau \multimap \rho \times \sigma \\
&\mathsf{fst}_{\rho\sigma} \;: \rho \times \sigma \multimap \rho \\
&\mathsf{snd}_{\rho\sigma} : \rho \times \sigma \multimap \sigma
\end{aligned}
$$

Terms are built from these constants and typed variables x^σ (incomplete variables) and $\bar{x}^\sigma$ (complete variables) by introduction and elimination rules for the two type forms $\rho \multimap \sigma$ and $\rho \to \sigma$, i.e.

$$
\begin{aligned}
&c^\rho \quad \text{(constant)} \mid \\
&x^\rho \quad \text{(incomplete variable)} \mid \\
&\bar{x}^\rho \quad \text{(complete variable)} \mid \\
&(\lambda x^\rho r^\sigma)^{\rho\multimap\sigma} \mid \\
&(r^{\rho\multimap\sigma} s^\rho)^\sigma \quad \text{with higher type incomplete variables in } r, s \text{ distinct} \mid \\
&(\lambda \bar{x}^\rho r^\sigma)^{\rho\to\sigma} \mid \\
&(r^{\rho\to\sigma} s^\rho)^\sigma \quad \text{with } s \text{ complete}
\end{aligned}
$$

We say that a term is *linear* or *ground* according as its type is. A term s is *complete* if all of its free variables are complete; otherwise it is *incomplete*. By the restriction on incomplete variables in the formation of (rs), a given higher type incomplete variable can occur at most once in a given term. We use infix notation for $*$, writing $l * r$ instead of $*rl$.

4. Conversions

The conversion rules are as expected.

$(\lambda x r)s$	$\mapsto r[x := s]$	β-conversion; similar for $\bar{x}$
$\mathsf{if}_\tau \mathsf{tt} s$	$\mapsto \mathsf{fst}_{\tau\tau} s$	
$\mathsf{if}_\tau \mathsf{ff} s$	$\mapsto \mathsf{snd}_{\tau\tau} s$	
$\boldsymbol{c}^\rho_\tau \varepsilon_\rho s$	$\mapsto \mathsf{fst}_{\tau,\sigma} s$	for $\sigma := \rho \multimap \mathbf{L}(\rho) \multimap \tau$
$\boldsymbol{c}^\rho_\tau (l *_\rho r) s$	$\mapsto \mathsf{snd}_{\tau,\sigma} srl$	for $\sigma := \rho \multimap \mathbf{L}(\rho) \multimap \tau$
$\mathcal{R}^\rho_\tau \varepsilon_\rho s t$	$\mapsto t$	
$\mathcal{R}^\rho_\tau (l *_\rho r) s t$	$\mapsto srl(\mathcal{R}^\rho_\tau l s t)$	
$\otimes^-_{\rho\sigma\tau}(\otimes^+_{\rho\sigma} r s) t$	$\mapsto trs$	

$$\mathsf{fst}_{\rho\sigma}(\times^{+}_{\rho\sigma\tau} rst) \;\mapsto rt$$
$$\mathsf{snd}_{\rho\sigma}(\times^{+}_{\rho\sigma\tau} rst) \mapsto st$$

Notice that we shall work with a representation of terms via parse dags, to be explained in Section 6. This will ensure that β-conversion of $(\lambda xr)s$ or $(\lambda \bar{x}r)s$ leads to a term within our system (cf. the proof of Lemma 6.2).

Redexes are subterms shown on the left side of conversion rules above. We assume that no two bound variables have the same name, and no bound variable has the same name as a free variable. A term is in *normal form* if it does not contain a redex. Write $r \to s$ for the one-step reduction based on the conversion rules, and $r \to^* s$ for its reflexive transitive closure. For every term t there is a unique normal-form term $\mathsf{nf}(t)$ such that $t \to^* \mathsf{nf}(t)$. Two terms are called *equivalent* if they have the same normal form.

5. Examples for exponential growth, again

We now come back to the examples form Section 2, and explain how our restrictions on the formation of types and terms make it impossible to build the corresponding terms. However, for definiteness we first have to say precisely what we mean by a numeral.

5.1. NUMERALS

Terms of the form $(\dots(\varepsilon_\rho *_\rho r_n^\rho)\dots *_\rho r_2^\rho) *_\rho r_1^\rho$ are called *lists*. We will make use of the following abbreviations for $\mathbf{N} := \mathbf{L}(\mathbf{U})$ and $\mathbf{W} := \mathbf{L}(\mathbf{B})$.

$$\begin{aligned}
0 &:= \varepsilon_{\mathbf{U}} \\
\mathsf{S} &:= \lambda l^{\mathbf{N}} l * \mathsf{xx} \\
1 &:= \varepsilon_{\mathbf{B}} \\
\mathsf{S}_0 &:= \lambda l^{\mathbf{W}} l * \mathsf{ff} \\
\mathsf{S}_1 &:= \lambda l^{\mathbf{W}} l * \mathsf{tt} \\
\mathsf{id} &:= \lambda x.x
\end{aligned}$$

Particular lists are $\mathsf{S}(\dots(\mathsf{S}0)\dots)$ and $\mathsf{S}_{i_1}(\dots(\mathsf{S}_{i_n}1)\dots)$. The former are called *unary numerals*, and the latter *binary numerals* (or *numerals of type* $\mathbf{W}$). We denote binary numerals by ν.

5.2. TWO RECURSIONS

Recall that we considered the definition of

$$\begin{array}{cc} d(1) := \mathsf{S}_0(1) & e(1) := 1 \\ d(\mathsf{S}_i(x)) := \mathsf{S}_0(\mathsf{S}_0(d(x))) & e(\mathsf{S}_i(x)) := d(e(x)) \end{array}$$

The corresponding terms are

$$\begin{aligned} d &:= \lambda\bar{x}.\mathcal{R}_{\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}}.\mathsf{S}_0(\mathsf{S}_0 p))(\mathsf{S}_0 1), \\ e &:= \lambda\bar{x}.\mathcal{R}_{\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}}.dp)1. \end{aligned}$$

Here d is legal, but e is not: the application dp is not allowed.

5.3. RECURSION WITH PARAMETER SUBSTITUTION

Recall the proposed definition of

$$\begin{array}{cc} e(1,y) := \mathsf{S}_0(y) & e(1) := \mathsf{S}_0 \\ e(\mathsf{S}_i(x),y) := e(x,e(x,y)) \quad \text{or} & e(\mathsf{S}_i(x)) := e(x) \circ e(x) \end{array}$$

The corresponding term

$$\lambda\bar{x}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}\multimap\mathbf{W}}\lambda y.p(py))\mathsf{S}_0$$

does not satisfy the linearity condition: the variable p occurs twice, and p needs to be incomplete because the type of $\mathcal{R}$ uses $\cdots \to (\rho \to \mathbf{L}(\rho) \to \tau \multimap \tau)$ rather than $\ldots \multimap (\rho \to \mathbf{L}(\rho) \to \tau \multimap \tau)$. Under the term formation rules, composition using such a type can only be with a complete term.

In our second example involving recursion with parameter substitution we had

$$\begin{array}{cc} e(1,y) := y & e(1) := \mathsf{id} \\ e(\mathsf{S}_i(x),y) := e(x,d(y)) \quad \text{or} & e(\mathsf{S}_i(x)) := e(x) \circ d \end{array}$$

The corresponding term would be

$$\lambda\bar{x}.\mathcal{R}_{\mathbf{W}\to\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}\to\mathbf{W}}\lambda\bar{x}.p(d\bar{x}))(\lambda yy)$$

but it is not legal, since the result type is not linear.

We considered

$$\begin{array}{rl} I(1,f,y) := y & \\ I(\mathsf{S}_i(x),f,y) := f(I(x,f,y)) & \text{or} \end{array} \qquad \begin{array}{rl} I(1,f) := \mathsf{id} & \\ I(\mathsf{S}_i(x),f) := f \circ I(x,f) & \end{array}$$

with the corresponding term

$$\begin{aligned} I_f &:= \lambda\bar{x}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}\multimap\mathbf{W}}\lambda y.f(py))(\lambda yy) \\ e &:= \lambda x.I_d x\mathsf{1} \end{aligned}$$

Here I_f is legal, but e is not: the type of d prohibits iteration. – Note that in PV^ω (Cook and Kapron (1990), Cook (1992)) I is *not* definable, for otherwise we could define $\lambda z.Idz$.

A related phenomenon occurs in

$$\begin{aligned} e(\mathsf{1}) &:= \mathsf{S}_0 \\ e(\mathsf{S}_i(x)) &:= I(\mathsf{S}_0(\mathsf{S}_0(\mathsf{1})), e(x)) \end{aligned}$$

Now the terms are

$$\begin{aligned} I_f &:= \lambda x.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}\multimap\mathbf{W}}\lambda y.f(py))(\lambda yy) \\ e &:= \lambda\bar{x}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}q^{\mathbf{W}\multimap\mathbf{W}}.I_q(\mathsf{S}_0(\mathsf{S}_0\mathsf{1})))\mathsf{S}_0 \end{aligned}$$

Again e is not legal, this time because the free parameter f in the step term of I_f is substituted with the *incomplete* variable q. This variable needs to be complete because of the typing of the recursion operator.

5.5. POLYNOMIALS

It is high time that we give some examples of what *can* de done in our term system. It is easy to define $\oplus\colon \mathbf{W} \to \mathbf{W} \multimap \mathbf{W}$ such that $x \oplus y$ concatenates $|x|$ bits onto y.

$$\begin{aligned} \mathsf{1} \oplus y &= \mathsf{S}_0 y \\ (\mathsf{S}_i x) \oplus y &= \mathsf{S}_0(x \oplus y) \end{aligned}$$

The representing term is $\bar{x} \oplus y := \mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda\bar{z}\lambda\bar{l}\lambda p^{\mathbf{W}\multimap\mathbf{W}}\lambda y.\mathsf{S}_0(py))\mathsf{S}_0$.

Similarly we define $\odot\colon \mathbf{W} \to \mathbf{W} \to \mathbf{W}$ such that $x \odot y$ has output length $|x| \cdot |y|$.

$$\begin{aligned} x \odot \mathsf{1} &= x \\ x \odot (\mathsf{S}_i y) &= x \oplus (x \odot y) \end{aligned}$$

The representing term is $\bar{x} \odot \bar{y} := \mathcal{R}_{\mathbf{W}} \bar{y} (\lambda \bar{z} \lambda \bar{l} \lambda p^{\mathbf{W}} \bar{x} \oplus p) \bar{x}$.

Notice that the typing $\oplus \colon \mathbf{W} \to \mathbf{W} \multimap \mathbf{W}$ is crucial: it allows using the incomplete variable p in the definition of $\odot$. If we try to go on and define exponentiation from multiplication $\odot$ just as $\odot$ was defined from $\oplus$, we find out that we cannot go ahead, because of the different typing $\odot \colon \mathbf{W} \to \mathbf{W} \to \mathbf{W}$.

6. Normalization

A *dag* is a directed acyclic graph. A *parse dag* is a structure like a parse tree but admitting in-degree greater than one. For example, a parse dag for $\lambda x r$ has a node containing λx and a pointer to a parse dag for r. A parse dag for (rs) has a node containing a pair of pointers, one to a parse dag for r and the other to a parse dag for s. Terminal nodes are labeled by constants and variables.

The *size* $|d|$ of a parse dag d is the number of nodes in it, but counting 3 for $\boldsymbol{c}_\tau$ nodes. Starting at any given node in the parse dag, one obtains a term by a depth-first traversal; it is the term *represented* by that node. We may refer to a node as if it were the term it represents.

A parse dag is *conformal* if (i) every node having in-degree greater than 1 is of ground type, and (ii) every maximal path to a bound variable x passes through the same binding λx node.

A parse dag is *h-affine* if each higher-type variable occurs at most once in the dag.

We adopt a model of computation over parse dags in which operations such as the following can be performed in unit time: creation of a node given its label and pointers to the sub-dags; deletion of a node; obtaining a pointer to one of the subsidiary nodes given a pointer to an interior node; conditional test on the type of node or on the constant or variable in the node. Concerning computation over terms (including numerals), we use the same model and identify each term with its parse tree. Although not all parse dags are conformal, every term is conformal (assuming a relabeling of bound variables).

A term is called *simple* if all its higher-type variables are incomplete. Obviously simple terms are closed under reductions, taking of subterms, and applications. Every simple term is h-affine, due to the linearity of incomplete higher-type variables.

Lemma 6.1 (Simplicity). *Let t be a ground type term whose free variables are of ground type. Then $\mathsf{nf}(t)$ contains no higher type complete variables.*

Proof. Let t be a ground type term whose free variables are of ground type, and consider $\mathsf{nf}(t)$. We must show that $\mathsf{nf}(t)$ contains no higher type

complete variables. So suppose a variable $\bar{x}^\sigma$ with $l(\sigma) > 0$ occurs in $\mathsf{nf}(t)$. It must be bound in a subterm $(\lambda \bar{x}^\sigma r)^{\sigma\to\tau}$ of $\mathsf{nf}(t)$. By the well known subtype property of normal terms, the type $\sigma \to \tau$ either occurs positively in the type of $\mathsf{nf}(t)$, or else negatively in the type of one of the constants or free variables of $\mathsf{nf}(t)$. The former is impossible since t is of ground type, and the latter by inspection of the types of the constants. □

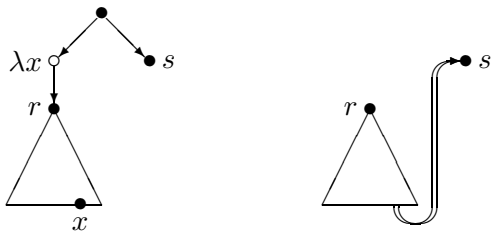
A term is $\mathcal{R}$-*free* if it does not contain an occurrence of $\mathcal{R}$.

Lemma 6.2 (Sharing Normalization). *Let t be an $\mathcal{R}$-free simple term. Then a parse dag for $\mathsf{nf}(t)$, of size at most $|t|$, can be computed from t in time $O(|t|^2)$.*

Proof. Under our model of computation, the input t is a parse tree. Since t is simple, it is an h-affine conformal parse dag of size at most $|t|$. If there are no nodes which represent a redex, then we are done. Otherwise, locate a node representing a redex; this takes time at most $O(|t|)$. We show how to update the dag in time $O(|t|)$ so that the size of the dag has strictly decreased and the redex has been eliminated, while preserving conformality. Thus, after at most $|t|$ iterations the resulting dag represents the normal-form term $\mathsf{nf}(t)$. The total time therefore is $O(|t|^2)$.

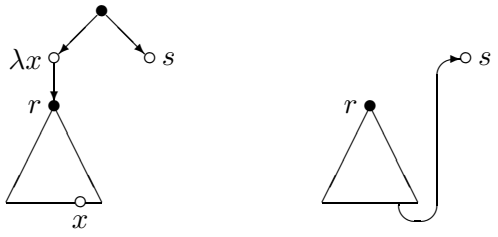
Assume first that the redex in t is $(\lambda x r)s$ with x of ground type; the argument is similar for a complete variable $\bar{x}$. Replace pointers to x in r by pointers to s. Since s does not contain x, no cycles are created. Delete the λx node and the root node for $(\lambda x r)s$ which points to it. By conformality (i) no other node points to the λx node. Update any node which pointed to the deleted node for $(\lambda x r)s$, so that it now points to the revised r subdag. This completes the β reduction on the dag (one may also delete the x nodes). Conformality (ii) gives that the updated dag represents a term t' such that $t \to t'$.

One can verify that the resulting parse dag is conformal and h-affine, with conformality (i) following from the fact that s has ground type.

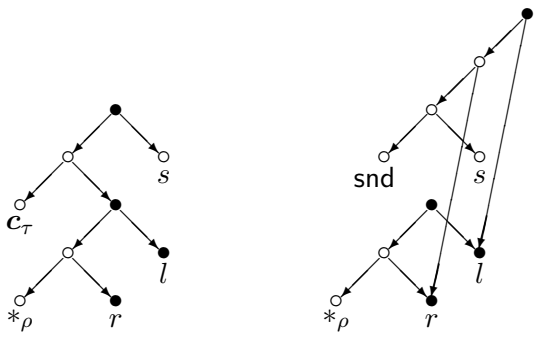


If the redex in t is $(\lambda x r)s$ with x of higher type, then x occurs at most once in r because the parse dag is h-affine. By conformality (i) there

is at most one pointer to that occurrence of x. Update it to point to s instead, deleting the x node. As in the preceeding case, delete the λx and the $(\lambda xr)s$ node pointing to it, and update other nodes to point to the revised r. Again by conformality (ii) the updated dag represents t' such that $t \to t'$. Conformality and acyclicity are preserved, observing this time that conformality (i) follows because there is at most one pointer to s.

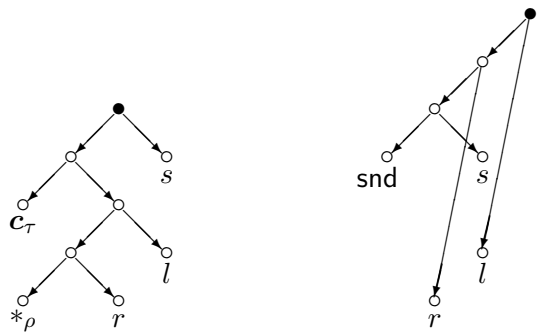


The remaining reductions are for the constant symbols.
Case $\mathsf{if}_\tau \mathsf{tt} s \mapsto \mathsf{fst}_{\tau\tau} s$. Easy; similar for ff.
Case $\boldsymbol{c}_\tau \varepsilon_\rho s \mapsto \mathsf{fst}\, s$. Easy.
Case $\boldsymbol{c}_\tau (l *_\rho r) s \mapsto \mathsf{snd}\, srl$ with ρ a ground type.

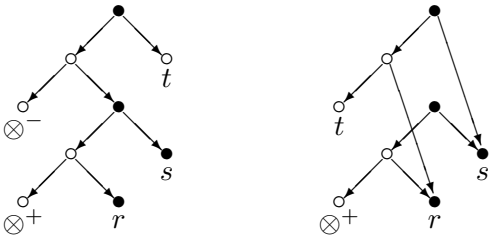


Notice that the new dag has one node more than the original one, but one $\boldsymbol{c}_\tau$-node less. Since we count the $\boldsymbol{c}_\tau$-nodes 3-fold, the total number of nodes decreases.

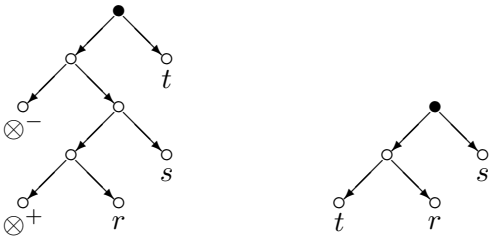
Case $\boldsymbol{c}_\tau(l *_\rho r)s \mapsto \mathsf{snd}\, srl$ with ρ not a ground type.



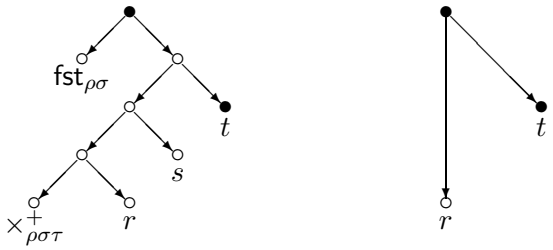
Case $\otimes^-_{\rho\sigma\tau}(\otimes^+_{\rho\sigma} rs)t \mapsto trs$ with $\rho \otimes \sigma$ a ground type.



Case $\otimes^-_{\rho\sigma\tau}(\otimes^+_{\rho\sigma} rs)t \mapsto trs$ with $\rho \otimes \sigma$ not a ground type.



Case $\mathsf{fst}_{\rho\sigma}(\times^+_{\rho\sigma\tau} rst) \mapsto rt$. Here we need that $\rho \times \sigma$ is *never* considered as a ground type. The case of $\mathsf{snd}_{\rho\sigma}(\times^+_{\rho\sigma\tau} rst) \mapsto st$ is of course similar.



□

Corollary 6.3 (Base Normalization). *Let t be a closed $\mathcal{R}$-free simple term of type* $\mathbf{W}$*. Then the binary numeral* $\mathsf{nf}(t)$ *can be computed from t in time $O(|t|^2)$, and $|\mathsf{nf}(t)| \leq |t|$.*

Proof. By Sharing Normalization (Lemma 6.2) we obtain a parse dag for $\mathsf{nf}(t)$ of size at most $|t|$, in time $O(|t|^2)$. Since $\mathsf{nf}(t)$ is a binary numeral, there is only one possible parse dag for it – namely, the parse tree of the numeral. This is identified with the numeral itself in our model of computation. □

Lemma 6.4 ($\mathcal{R}$ Elimination). *Let t be a simple term of linear type. There is a polynomial p_t such that: if $\vec{m}$ are linear type $\mathcal{R}$-free closed simple terms and the free variables of $t[\vec{x} := \vec{m}]$ are linear and incomplete, then in time $p_t(|\vec{m}|)$ one can compute an $\mathcal{R}$-free simple term $\mathsf{rf}(t; \vec{x}; \vec{m})$ such that $t[\vec{x} := \vec{m}] \to^* \mathsf{rf}(t; \vec{x}; \vec{m})$.*

Proof. By induction on $|t|$.

If t has the form $\lambda z u_1$, then z is incomplete and z, u_1 have linear type because t has linear type. If t is of the form $D\vec{u}$ with D a variable or one of the symbols xx, tt, ff, ε_ρ, $*_\rho$, if_τ, $\boldsymbol{c}_\tau$, $\otimes^+_{\rho\sigma}$, $\otimes^-_{\rho\sigma\tau}$, $\times^+_{\rho\sigma\tau}$, $\mathsf{fst}_{\rho\sigma}$ or $\mathsf{snd}_{\rho\sigma}$, then each u_i is a linear type term. Here (in case D is a variable x_i) we need that x_i is linear.

In all of the preceeding cases, each $u_i[\vec{x} := \vec{m}]$ has only linear and incomplete free variables. Apply the induction hypothesis as required to simple terms u_i to obtain $u_i^* := \mathsf{rf}(u_i; \vec{x}; \vec{m})$; so each u_i^* is $\mathcal{R}$-free. Let t^* be obtained from t by replacing each u_i by u_i^*. Then t^* is an $\mathcal{R}$-free simple term; here we need that $\vec{m}$ are closed, to avoid duplication of variables. The result is obtained in linear time from $\vec{u}^*$. This finishes the lemma in all of these cases.

If t is $(\lambda y r)s\vec{u}$ with an incomplete variable y of ground type, apply the induction hypothesis to yield $(r\vec{u})^* := \mathsf{rf}(r\vec{u}; \vec{x}; \vec{m})$ and $s^* := \mathsf{rf}(s; \vec{x}; \vec{m})$. Redirect the pointers to y in $(r\vec{u})^*$ to point to s^* instead. If t is $(\lambda \bar{y} r)s\vec{u}$ with a complete variable $\bar{y}$ of ground type, apply the IH to yield $s^* := \mathsf{rf}(s; \vec{x}; \vec{m})$. Notice that s^* is closed, since it is complete and the free variables of $s[\vec{x} := \vec{m}]$ are incomplete. Then apply the induction hypothesis again to obtain $\mathsf{rf}(r\vec{u}; \vec{x}, \bar{y}; \vec{m}, s^*)$. The total time is at most $q(|t|) + p_s(|\vec{m}|) + p_r(|\vec{m}| + p_s(|\vec{m}|))$, as it takes at most linear time to construct $r\vec{u}$ from $(\lambda y r)s\vec{u}$.

If t is $(\lambda y r)s\vec{u}$ with y of higher type, then y can occur at most once in r, because t is simple. Thus $|r[y := s]\vec{u}| < |(\lambda y r)s\vec{u}|$. Apply the induction hypothesis to obtain $\mathsf{rf}(r[y := s]\vec{u}; \vec{x}; \vec{m})$. Note that the time is bounded by $q(|t|) + p_{r[y:=s]\vec{u}}(|\vec{m}|)$ for a degree one polynomial q, since it takes at most linear time to make the at-most-one substitution in the parse tree.

The only remaining case is if the term is an $\mathcal{R}$ clause. Then it is of the form $\mathcal{R}lst\vec{}$, because the term has linear type.

Since l is complete, all free variables of l are complete – they must be in $\vec{x}$ since free variables of $(\mathcal{R}lst\vec{})[\vec{x} := \vec{m}]$ are incomplete. Then $l[\vec{x} := \vec{m}]$ is closed, implying $\mathsf{nf}(l[\vec{x} := \vec{m}])$ is a list. One obtains $\mathsf{rf}(l; \vec{x}; \vec{m})$ in time $p_l(|\vec{m}|)$ by the induction hypothesis. Then by Base Normalization (Corollary 6.3) one obtains the list $\hat{l} := \mathsf{nf}(\mathsf{rf}(l; \vec{x}; \vec{m}))$ in a further polynomial time. Let $\hat{l} = (\dots(\varepsilon_\rho *_\rho r_N)\dots *_\rho r_1) *_\rho r_0$ and let l_i, $0 \leq i \leq N$ be obtained from $\hat{l}$ by omitting the initial elements $r_0, \dots, r_i$. Thus all $\{\, r_i, l_i \mid i \leq N \,\}$ are obtained in a total time bounded by $p'_l(|\vec{m}|)$ for a polynomial p'_l.

Now consider $s\bar{y}\bar{z}$ with new variables $\bar{y}^\rho$ and $\bar{z}^{\mathbf{L}(\rho)}$. Applying the induction hypothesis to $s\bar{y}\bar{z}$ one obtains a monotone bounding polynomial $p_{s\bar{y}\bar{z}}$. One computes all $s_i := \mathsf{rf}(s\bar{y}\bar{z}; \vec{x}, \bar{y}, \bar{z}; \vec{m}, r_i, l_i)$ in a total time of at most

$$\sum_{i=1}^{N} p_{s\bar{y}\bar{z}}(|r_i| + |l_i| + |\vec{m}|) \leq p'_l(|\vec{m}|) \cdot p_{s\bar{y}\bar{z}}(2p'_l(|\vec{m}|) + |\vec{m}|).$$

Each s_i is $\mathcal{R}$-free by the induction hypothesis. Furthermore, no s_i has a free incomplete variable: any such variable would also be free in s contradicting that s is complete.

Consider $\vec{t}$. The induction hypothesis gives all $\hat{t}_i := \mathsf{rf}(t_i; \vec{x}; \vec{m})$ in time $\sum_i p_{t_i}(|\vec{m}|)$. These terms are also $\mathcal{R}$-free by induction hypothesis. Clearly the t_i do not have any free (or bound) higher type incomplete variables in common. The same is true of all $\hat{t}_i$.

Using additional time bounded by a polynomial p in the lengths of these computed values, one constructs the $\mathcal{R}$-free term

$$(\lambda z.s_0(s_1 \dots (s_N z)\dots))\vec{\hat{t}}.$$

Defining $p_t(n) := p(\sum_i p_{t_i}(n) + p'_l(n) \cdot p_{syz}(2p'_l(n) + n))$, the total time used in this case is at most $p_t(|\vec{m}|)$. The result is a term because the $\hat{t}_i$ are terms which do not have any free higher-type incomplete variable in common and because s_i does not have any free higher-type incomplete variables at all. □

7. Characterizations

Lemma 7.1 (Normalization). *Let t be a closed term of type $\vec{\rho} \multimap \sigma$, where σ and each ρ_i is a ground type. Then t denotes a polytime function.*

Proof. One must find a polynomial q_t such that for all $\mathcal{R}$-free simple closed terms $\vec{n}$ of types $\vec{\rho}$ one can compute $\mathsf{nf}(t\vec{n})$ in time $q_t(|\vec{n}|)$. Let $\vec{x}$ be new

variables of types $\vec{\rho}$. The normal form of $t\vec{x}$ is computed in an amount of time that may be large, but it is still only a constant with respect to $\vec{n}$.

$\mathsf{nf}(t\vec{x})$ is simple by Lemma 6.1. By $\mathcal{R}$ elimination (Lemma 6.4) one reduces to an $\mathcal{R}$-free simple term $\mathsf{rf}(\mathsf{nf}(t\vec{x}); \vec{x}; \vec{n})$ in time $p_t(|\vec{n}|)$. Since the running time bounds the size of the produced term, $|\mathsf{rf}(\mathsf{nf}(t\vec{x}); \vec{x}; \vec{n})| \leq p_t(|\vec{n}|)$.

By Sharing Normalization (Lemma 6.2) one can compute $\mathsf{nf}(t\vec{n}) = \mathsf{nf}(\mathsf{rf}(\mathsf{nf}(t\vec{x}); \vec{x}; \vec{n}))$ in time $O(p_t(|\vec{n}|)^2)$. Let q_t be the polynomial referred to by the big-O notation. □

Lemma 7.2 (Sufficiency). *Let f be a polynomial-time computable function of type $\vec{\mathbf{W}} \multimap \mathbf{W}$. Then f is denoted by a closed term t.*

Proof. In Bellantoni and Cook (1992) the polynomial time computable functions are characterized by a function algebra B based on *safe recursion* and *safe composition*. There every function is written in the form $f(\vec{x}; \vec{y})$ where $\vec{x}; \vec{y}$ denotes a bookkeeping of those variables $\vec{x}$ that are used in a recursion defining f, and those variables $\vec{y}$ that are not recursed on. We proceed by induction on the definition of $f(x_1, \ldots, x_k; y_1, \ldots, y_l)$ in B, associating to f a closed term t_f of type $\mathbf{W}^{(k)} \to \mathbf{W}^{(l)} \multimap \mathbf{W}$, such that t denotes f.

The functions in B were defined over the non-negative integers rather than the positive ones, but this clearly is a minor point. We use the bijection $x \in \mathbb{N} \Leftrightarrow (2^{|x|} + x) \in \mathbb{Z}^+$.

If f in B is an initial function 0, S_0, S_1, P, conditional C or projection $\pi_i^{m,n}$, then t_f is easily defined.

If f is defined by safe composition in system B, then

$$f(\vec{x}; \vec{y}) := g(r_1(\vec{x};), \ldots, r_m(\vec{x};); s_1(\vec{x}; \vec{y}), \ldots, s_n(\vec{x}; \vec{y})).$$

Using the induction hypothesis to obtain t_g, $t_{\vec{r}}$ and $t_{\vec{s}}$, define

$$t_f := \lambda\vec{\bar{x}}\lambda\vec{y}.t_g(t_{r_1}\vec{\bar{x}}) \ldots (t_{r_m}\vec{\bar{x}})\,(t_{s_1}\vec{\bar{x}}\vec{y}) \ldots (t_{r_m}\vec{\bar{x}}\vec{y}).$$

Finally consider f defined by safe recursion in system B.

$$\begin{aligned} f(0, \vec{x}; \vec{y}) &:= g(\vec{x}; \vec{y}) \\ f(\mathsf{S}_i n, \vec{x}; \vec{y}) &:= h_i(n, \vec{x}; \vec{y}, f(n, \vec{x}; \vec{y})) \quad \text{for } \mathsf{S}_i n \neq 0. \end{aligned}$$

One has t_g, t_{h_0} and t_{h_1} by the induction hypothesis. Let p be a variable of type $\tau := \mathbf{W}^{(\#(\vec{y}))} \multimap \mathbf{W}$; this is the linear type used in the recursion. Then define a step term by

$$s := \lambda\bar{x}\lambda\bar{l}\lambda p\lambda\vec{y}.\mathsf{if}_{\mathbf{W}}\bar{x}(\times^+(\lambda z.t_{h_0}\bar{l}\vec{\bar{x}}\vec{y}z)(\lambda z.t_{h_1}\bar{l}\vec{\bar{x}}\vec{y}z)(p\vec{y})).$$

Note p is used only once. Let $t_f := \lambda\bar{n}\lambda\vec{\bar{x}}.\mathcal{R}_\tau\bar{n}s(t_g\vec{\bar{x}})$. □

References

Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

Stephen Bellantoni and Martin Hofmann. A New "Feasible" Arithmetic *Journal of Symbolic Logic*, to appear.

Stephen Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.

Stephen A. Cook. Computability and complexity of higher type functions. In Y.N. Moschovakis, editor, *Logic from Computer Science, Proceedings of a Workshop held November 13–17, 1989*, number 21 in MSRI Publications, pages 51–72. Springer Verlag, Berlin, Heidelberg, New York, 1992.

Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 71–96. Birkhäuser, 1990.

Jean-Yves Girard. Light linear logic. *Information and Computation*, 143, 1998.

Jean-Yves Girard, Andre Scedrov, and Philipp J. Scott. Bounded linear logic. In S.R. Buss and Ph.J. Scott, editors, *Feasible Mathematics*, pages 195–209. Birkhäuser, Boston, 1990.

Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

David Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–190, 1925.

Martin Hofmann. Typed lambda calculi for polynomial-time computation. Habilitation thesis, Mathematisches Institut, TU Darmstadt, Germany. Available under www.dcs.ed.ac.uk/home/mxh/habil.ps.gz, 1998.

Daniel Leivant. Intrinsic theories and computational complexity. In *Logic and Computational Complexity, International Workshop LCC '94, Indianapolis* D. Leivant, ed., Springer LNCS 960, 1995, p. 177-194.

Daniel Leivant. Predicative recurrence in finite type. In A. Nerode and Y.V. Matiyasevich, editors, *Logical Foundations of Computer Science*, volume 813 of *LNCS*, pages 227–239, 1994.

Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity IV: Predicative functionals and poly-space. To appear: Information and Computation.

Daniel Leivant and Jean-Yves Marion. Lambda calculus characterization of poly–time. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, pages 274–288. LNCS Vol. 664, 1993.

Harold Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.